Coordinated Fault Tolerance for High Performance Computing

Peter Beckman, William Gropp, Ewing Lusk,
Robert Ross, and Rajeev Thakur

Mathematics and Computer Science Division
Argonne National Laboratory

Al Geist

Computer Science and Mathematics Division
Oak Ridge National Laboratory

December 1, 2005

1 Executive Summary

Ultra-scale platforms are compound hierarchical systems. Currently, the world's fastest computer uses over 100,000 CPUs for computation, 1000 systems for disk storage, and 500 nodes for I/O forwarding. The next version of this architecture will likely double those numbers, and the trend is not slowing - machines from every vendor are becoming more complex as they scale up.

The petascale applications that are evolving to utilize these platforms face many new challenges. The fault management issues for these emerging systems are well beyond the scope of today's common infrastructure and practice. Currently, systems software components for large-scale machines remain largely independent in their fault awareness and notification strategies. Faults can arise not just from the hardware but also from the OS, middleware, and application levels. Often, only the most rudimentary error conditions such as "job failed", "file write failed, or worse yet, a hung application provide scientists any notion that something has gone awry. Moreover, the multiple layers of software between application and computer have little to no opportunity to report, avoid, or correct the issue. Software fault detection and correction for supercomputers has remained largely unchanged for more than two decades.

Our vision is to provide application scientists with access to ultra-scale platforms that behave predictably and hide the occurrence of hardware and system software faults. Scientists running on such a machine would never be allocated resources that were not performing as expected, would never have to restart their job manually because of a hardware failure, and would never be charged node-hours for time wasted due to problems with the system. Instead, the machine would take underperforming resources off-line, delay scheduling of jobs when necessary resources are not

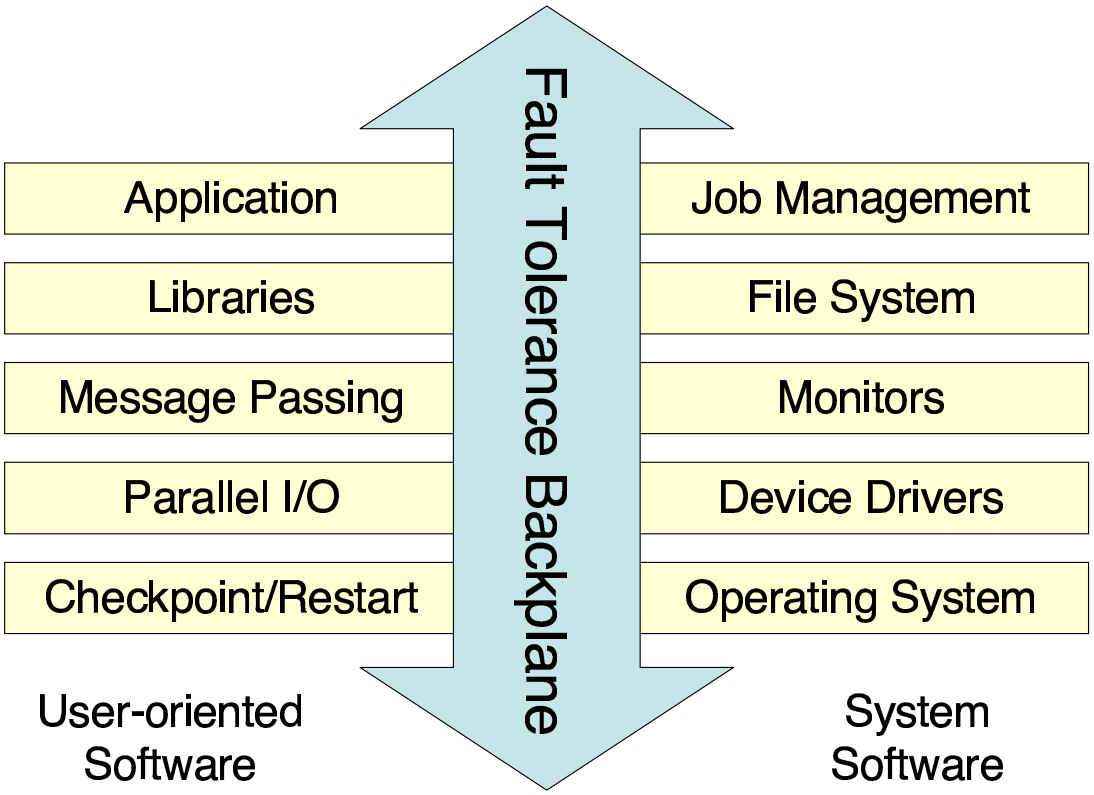

Figure 1: The Fault Tolerance Backplane will provide a shared infrastructure for user-oriented and system system to coordinate fault information and response

available, and restart jobs in the event of a system component failure.

We believe a coordinated approach to enable large systems to react to faults and interact with both applications and other system components is required to provide resilience for petascale applications and productivity gains for users. We propose to design and implement interfaces for integrating fault tolerance features from multiple layers of the system software stack, from the application and programming system layer through the file system and other parallel system software to the operating system. Such integration will make possible a level of fault notification, management, and recovery that is impossible for any single element to deliver.

In this project we will extend and adapt existing system software and middleware to plug into a "backplane" and enable systems to react to faults in a holistic manner. We will focus on three specific areas:

- Design and deployment of a fault awareness and notification backplane to provide common

uniform event handling and notification mechanisms for fault-aware libraries and middleware.

- System software tools and strategies to make existing libraries, run-time systems, and OS kernels fault aware, and connect to and use the fault tolerant backplane.
- User-level interfaces, tools, and methods for integrating applications into the fault tolerant backplane. Where possible we will leverage DOE projects in parallel libraries and languages, operating systems, file systems, and scalable systems software. We will also work with application teams to help them exploit this new technology.

2 Introduction

As systems are built from ever-larger numbers of components, fault detection, resiliency, and autonomic behavior become increasingly important. Large-scale machines such as IBM's Blue Gene and Cray's XT3 are designed and built with impressive hardware fault detection and recovery mechanisms. If a chip overheats, it is automatically turned off. If a network link gets CRC errors, an alert is raised. A redundant power supply can automatically switch on when another fails. Within the system software, operating system, middleware, and user code, however, very little work has been done to dynamically respond to fault detection. For example, if one of the storage nodes in a parallel file system gets a disk error, the user application usually simply aborts with a "write failed" error. Similarly, when a compute node fails, the most common behavior is for the message layer to simply time-out and the application to eventually fail. Even fault tolerant extensions to MPI generally only detect nodes that are no longer responding and then begin MPI-level mitigation strategies. Knowledge of alternate interfaces, network tuning parameters, processor remapping, and the status of failover disk storage lives completely outside these user-level communication libraries. The current generation of scalable middleware is completely unequipped to detect and adaptively respond to faults in ultra-scale environments because they do not take a holistic, full-system approach; fault information is isolated, and actions uncoordinated.

The Scalable Systems Software SciDAC project has set the stage by building infrastructure to share public interfaces and coordinate system software components for large machines. The result has been a set of system management components, all sharing a common communication library supporting multiple security levels and wire protocols, with public XML interfaces that allow any other component to access their assorted functions. Both synchronous communication (send/receive) and asynchronous communication (register/notify) are supported, allowing great flexibility in how components are configured. It is easy to incorporate powerful components written by others (such as alternative monitoring, scheduling, or accounting components) by embedding them in "wrappers" that communicate with other components using the public interfaces. The practicality of this approach has been demonstrated over the past year by operating Argonne's 256-processor "Chiba City" cluster and 2048 processor IBM Blue Gene in production mode for both application use and computer science research with system software consisting entirely of Scalable Systems Software components [1]. This collection of existing components provides one foundation on which we plan to build the fault-tolerance backplane.

2.1 A Backplane for Sharing and Coordinating Fault Information

Our innovative approach to addressing the need for fault management and autonomic computing begins with the backplane, the coordinating infrastructure that enable all levels of software to both share and respond to fault information. FOBAWS, the Fault Organizing Backplane for Autonomic WidgetS, will be designed and built to provide light-weight coordination. FOBAWS will let applications survive mistakes. FOBAWS will leverage existing libraries, run-time systems, and operating systems. Packages such as PVFS2, BLCR, MPICH, ROMIO, Cobalt, ZeptoOS, OpenMPI, and LAM-MPI all provide a component within the system software stack, and can all benefit from sharing an integrated component-based framework for detecting and propagating fault information. Beginning with this software, which spans operating systems, file systems, message passing layers, libraries, and job management interfaces, we will design interfaces and components that can be shared across all layers of scalable system software— controlling and responding to faults within the system. FOBAWS modular design is based on four major components: the collection system, the subscription and configuration layer, the event notification component, and the autonomic logic engine.

At the lowest level of the fault management and autonomic computing system are the data collectors that gather information about the basic system. This is where we intend to leverage packages already widely available within the open source community. Many systems exist that collect and display live system information. Inca, used by the TeraGrid, uses a standardized XML schema that enables many authors to write "reporters" to collect data and propagate that information to a centralized database. Similarly, Nagios, Clumon, Ganglia, and SNMP all provide mechanisms to periodically collect data from active machines. We intend to use these existing systems and collect information that can be the foundation for detecting and responding to anomalies in the system.

The subscription and configuration layers provide an up-to-date inventory of the system components. Such an inventory is required both for understanding faults and for responding to them. Are there any spare resources that can be retasked to take over? How many servers belong to the PVFS storage system? What job is currently associated with node 2? The SciDAC Scalable Systems Software work can be heavily leveraged in building this component. The Scalable Systems Software work has developed standardized interfaces for many configuration tasks, such as job management.

FOBAWS will present an event and callback interface through which basic data on system events can be communicated to and from user applications or fault-aware libraries (such as an MPI implementation). Each system component must be able to register "interest" in a variable or condition. For example, the OpenMPI software may want to be notified if one of the nodes is down or a network interface is receiving many bit-errors. The key innovation, however, is that FOBAWS will share that information with consistent interfaces across many levels of system software, from the parallel file system, PVFS, to the math libraries. Applications and middleware will then cooperate in taking appropriate action.

Finally, the autonomic logic engine will orchestrate the system response to faults. The logic engine provides a hierarchical set of execution sites where "plug-ins" can be executed and react to local information. With the backplane designed to be extremely light weight, it is the job of the plug-ins to coordinate action across components.

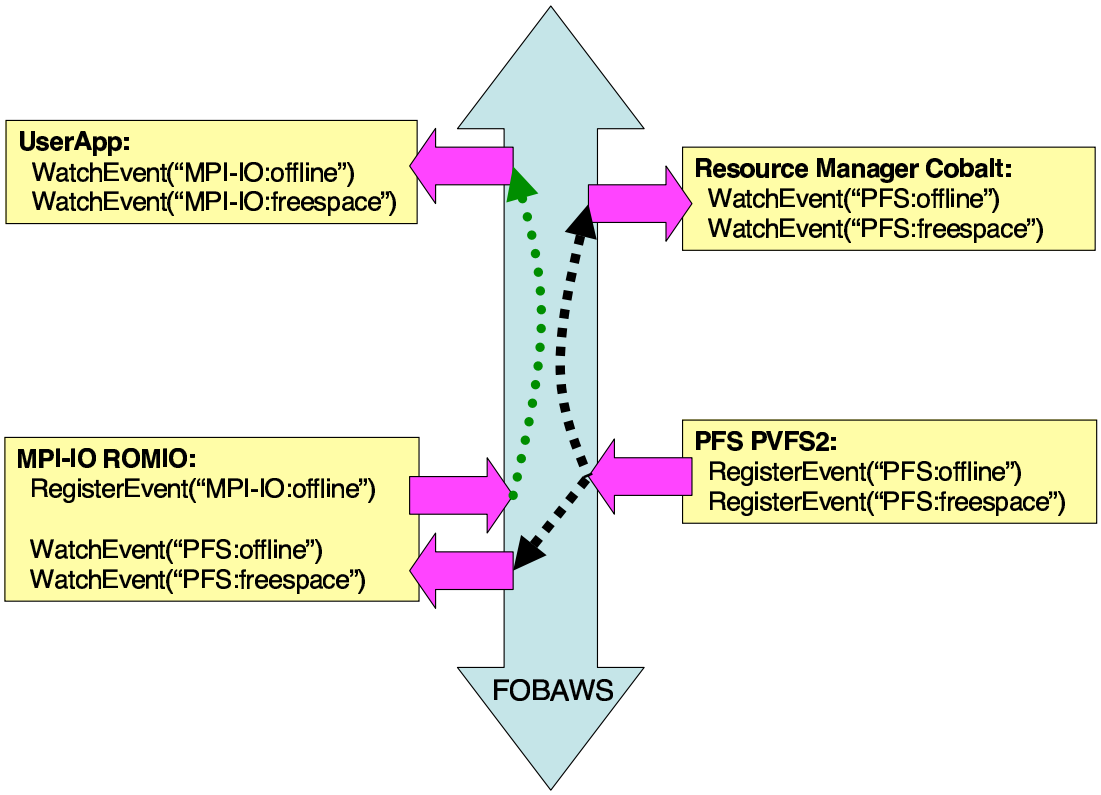

Figure 2: Scenario 1: The Parallel File System Becomes Unavailable

3 Usage Scenarios

To demonstrate how fault-aware components at all the levels of the software stack can interact and coordinate responses to faults in our design, we present four use-case scenarios. These examples not only motivated our design, but they will also be used as milestones for the project.

3.1 Scenario 1: Parallel Storage is Offline

Unlike home directory partitions, which predictably use increasingly more disk space until the administrator sends out a warning to the largest users, parallel file systems are linked to parallel jobs, and the available storage may fluctuate wildly based on which jobs have run recently, and which data sets have not been moved off to other storage. The volatile nature of parallel storage availability makes it an prime target for improved fault tolerance. As a concrete example, let's consider the fault-awareness interactions between MPI-IO (ROMIO), a parallel file system (PVFS2),

and a job scheduler and resource manager (Cobalt).

PVFS2, a parallel file system for HPC, uses a fixed set of storage servers to provide multiple data paths and increase throughput. Two common issues affect end-user application availability: free space and dead servers. With the current version of PVFS2, a dead server without a failover partner (a common deployment to reduce server costs) will make the entire distributed volume unavailble. It is also a condition that PVFS2 can easily determine within its own communication system, but is difficult for outside systems to properly detect. Using FOBAWS to share and then coordinate corrective action is ideal.

Figure 3.1 shows how components can register or watch events. Each component can decide how it should use event data. In our example, when Cobalt, the resource manager and job scheduler is notified that PVFS2 is offline or freespace dangerously low, it can hold all jobs in the queue that depend on PVFS2 and email users that their jobs will be held in the queue until PVFS2 service is restored. Likewise, MPI-IO, which can use PVFS2 may wish to be notified if the service is unavailable, and pass that information to the application. Applications that do not take full advantage of FOBAWS would simply be notified of the error when attempting to use MPI-IO. However, rather than having the file write hang indefinately or simply get an error that the file operation failed, a useful message could be sent to stderr that MPI-IO failed because PVFS2 is offline. Applications which choose to manage fault information could, for example, choose to turn off PVFS2 checkpointing and write its final output file to the slower, but functioning NFS server.

3.2 Scenario 2: A Cluster Node is Failing

There are many ways for a node to cause mischief. From a fault management perspective, nodes that suddenly die are the simplest to handle. More diffuclt, is when a flagging processor becomes slow or the interconnect generates rivers of networking errors that force message layer retransmissions. Nodes can also begin behaving badly because of run-amok services, such as a background process that starts hoarding memory and forces swapping. To address these issues we include the concept of a *performance fault*, when expected performance is so far below acceptable levels that the fault system should be activated, even though nodes may technically remain "up". The operating system, device drivers, and low level management and monitoring system must all cooperate with the HPC messaging layer to detect and signal faults.

The faults described above could be detected either by the middleware (MPI messages fail), or the a combination of the node's own low-level system software and external monitoring packages. However the fault is first detected, we will build an autonomic plug-in to coordinate a holistic, system-wide response. First, the application and other middleware layers must be notified. Our strategy is to enhance all systems using the backplane, even if end-suer applications remain oblivious to its existance. Faulty or even misbehaving nodes will be reported to the user application via the job management system software, even if the application is not fault aware. A simple message to standard error can announce the difficulty. This straightforward user-level response is much appreciated by users normally faced with deducing why their job failed. However, with the autonomic plug-in, much more can be accomplished.

Applications or middleware that are fault aware and respond to the backplane announcing a

critical fault can choose to continue execution. While the default behavior for HPC systems is to kill the entire job if one node fails, applications or fault-aware layers could overide this behaviour. An application built with a fault tolerant MPI can simply indicate that the job will keep running, even if nodes die. However, when an application has no other choice but to terminate, the autonomic engine is far from done.

Next, the user's CPU accounting must be refunded. If a job dies because of a node failure, a refund should follow. Also, if the application or middleware supports checkpoint/restart, the job will automatically resubmitted to the job queues, possibly at the head of the line. Furthermore, the system management system components must also do their part to mitigate the failure. Where there are spare nodes, they will be brought on line to replace the bogus node. Otherwire, the resource manager will mark the node "offline" and ensure jobs will not be scheduled to run on it. Finally, the autonomic engine will automatically save citical log files from the ailing node (if possible) and then launch a full diagnostic reboot and test regime to qualify the node before attempting to put it back into service.

3.3 Scenario 3: Checkpoint / Restart

This is where we cover both signalling an application to be frozen and rolled out (system initiated) and user-initialted checkpoint/restart

3.4 Scenario 4: Faulty Interconnection

collective faults, etc

4 The Areas

For each of these areas, we need to discuss: the challenges, and what can be achieved 1) separately and 2) as part of FABS.

4.1 MPI

An MPI implementation is in a good position both to recognize faults and to report them. An MPI library makes calls to lower-level communication libraries such as sockets, Myrinet, or Infiniband libraries supplied by OS or networks vendors, and those libraries will return error codes to the calling layer in the MI implementation. The MPI Standard specifies a highly flexible behavior for processing errors. According to the Standard, errors may cause the parallel job to abort, may cause error codes to be returned to the caller of the highest level MPI function, or invoke a user-defined error handler, at the discretion of the application or library making the MPI calls. Error handling behavior can also be customized for individual communicators.

In [2], we described an approach to fault tolerance that relies only on the MPI standard itself,

and outlined some of the ways in which an application can itself become more fault tolerant by taking advantage of some features of the MPI specification that would be present in any high-quality MPI implementation. However, in the presence of the fault-tolerance backplane, much more could be done. The MPI library, in addition to what it does to implement the Standard, could also notify the fault-tolerant backplane of whatever details it has learned from the failure of the communication layer. At that point any component that has registered to receive such notification would be notified and could take appropriate action. Such components might include a checkpoint manager, network monitor, file system monitor, or account system, each of which might be able to provide services to the application to minimize the effect of the fault.

4.2 PVFS

4.3 ROMIO

4.4 ZeptoOS

4.5 Cobalt

5 Technical Approach

Predicting Failures

Preventing system failures before they occur is the direct method for preventing application errors and system downtime. Many modern hardware systems are now collecting data that is useful for predictive failure analysis. Many failures are actually not sudden; hints of impending trouble provide clues before failure. Modern disk drives have special interfaces that can be used to probe heath statistics, such as spin-up time, head flying height, and bit error rates. ECC memory, network interfaces, and motherboard chassis also have vital reliability data that can be collected.

Unfortunately, current system software does not record information about impending faults or even maintain a history of which components are most likely to fail. To address this situation, our project will explore collecting software component failure rates as well as possible precursors to failure. The Linux kernel has already added a couple of these internal checks. For example, if the kernel detects that a process is spawning too rapidly, it guesses that a failure is happening, and adjusts the system. Collecting information from the scalable system software will enable the fault detection and autonomic system to predict and avoid application failure.

We propose to further port and develop this system to allow its use on a broad range of systems, including all those in the NLCF. The NLCF includes a diverse set of supercomputers: Linux clusters at ANL, ORNL, and PNNL; an IBM BG/L at ANL; and a Cray X1E and Cray XT3 at ORNL. Development often starts on more personal systems. As such, we will also port this environment to laptops and desktop computers.

Backplane

o Registration of Interest

o Emit Event

o communication layer

o monitors/autonomic components/watchdogs

o Future fault event (even scheduled, such as preventive maint)

o scalability of system

What events?

o schema (SNMP)

o performance faults

o disk, net

o condor resource description language as reference?

6 Milestones

Year 1 Getting started

- Tell users that abort will happen
- User (i.e., middleware lib) tells system failure happened
- define interfaces, build prototype for event system

Year 2 Initial example scenarios

- Task farm example (Steve Pieper's QMC code)
- Libraries can event/consume to each other

Year 3 Initial releases of backplane aware instances of components

- Job requirements and current state of errors affect scheduler
- All participants release Software to support FOBAWS

Year 4 Usage with applications

- KeLP restart from checkpoint with hooks to scheduler
- Scalability testing

Year 5 Advanced features into released software

- Toolkit for watchdogs, autonomic components, etc.
- L.C. software

7 Why Us?

Why is this group the best one to undertake this project? Four particular reasons are 1) immediate background, 2) familiarity with instances of major components, 3) record of delivering useful software, and 4) access to actual systems to develop and test on.

1. In some ways this project is a continuation of the Scalable Systems Software SciDAC. It has a tighter focus and a smaller cast of characters, but it continues that project's emphasis on the specification of open, public interfaces that allow multiple instantiations of individual components to use the common infrastructure. Lessons learned there about portable, scalable interface defintions will be applied in this project.

2. This group encompasses implementors of *instances* of many of the key components: MPI implementation library (MPICH2, OpenMPI), parallel file system (PVFS2), operating system for individual nodes (ZeptoOS), configuration system (BCFG), checkpoint manager (BCLR) and others. This will allow us to test our backplane ideas in the context of state-of-the-art instances of the various components. Because of the openness of the planned architecture, of course, other versions of such components will also be able to make use of the fault tolerance backplane. In order to ensure this in the case of the MPI implementation in particular, we have included in the group two different open-source MPI implementations.

3. This group has a track record of transitioning research software into real community software in wide use on a variety of platforms. MPICH, Cobalt, and PVFS are just three examples.

4. We have at our disposal several research and production systems to develop and test on. The Radix Lab at Argonne includes Chiba City, a 512-cpu system that routinely runs a mix of research and production jobs; we also intend to experiment on the BG/L system a Argonne and the leadership class machines at Oak Ridge.

8 Budget Summary

9 Summary

References

[1] N. Desai, R. Bradshaw, A. Lusk, E. Lusk, and R Butler. Component-based cluster systems software architecture: A case study. In *IEEE International Conference on Cluseter Computing*, 2004.

[2] William D. Gropp and Ewing Lusk. Fault tolerance in MPI programs. *International Journal of High Performance Computer Applications*, 18(3):363–372, 2004.

A Appendix: Any Appendix here

B Appendix: Any other Appendix here

C Appendix: Letters of Support